TALLINNA ÜLIKOOL
Haapsalu kolledž
Rakendusinformaatika õppekava

Taaniel Levin

RAKENDUSLIIDESTE REGRESSIOONITESTIMISE AUTOMATISEERIMINE

Diplomitöö

Juhendaja: MSc Andrus Rinde

Haapsalu 2023

SISUKORD

SISSEJUHATUS

Rakendusliideste testimine on aktuaalne teema, kuna testimine on tarkvaraarenduse kvaliteedi tagamisel väga oluline osa. Tarkvara peaks olema töökindel ning veavaba, et lõppkasutajal tekiks tarkvara kasutamisel võimalikult vähe probleeme. Suurtes arendusprojektides on tähtis, et lisaks manuaalsele testimisele oleks olemas ka hästi koostatud automaattestide süsteem, mis tagab varem arendatud osa kiire testimise ja aitab tagada selle funktsioneerimise ka peale uusi muudatusi. Viimasel ajal luuakse üha rohkem tarkvararakendusi, mis integreeruvad teiste rakendustega, platvormidega ja süsteemidega. Rakendusliides on tarkvara komponent, mis võimaldab erinevatel rakendustel omavahel suhelda ja andmeid vahetada. Kuna rakendused muutuvad üha keerukamaks ja sisaldavad aina rohkem rakendusliideseid, on oluline tagada nende liideste usaldusväärsus ja tõrgeteta toimimine. Rakendusliideste testimine on oluline tagamiseks, et erinevad rakendused suhtlevad omavahel õigesti ja tõrgeteta ning et andmeid vahetatakse turvaliselt ja õiges vormingus.

Sellist tüüpi testimist, mille eesmärk on veenduda, et uued koodimuudatused ei kahjusta juba eksisteerivat arendust, nimetatakse regressioonitestimiseks. See kindlustab kogu tarkvara toimimise ning on efektiivne vahend hoidmaks ära vigu tarkvaras (Software Testing Help, 2022). Regressioonitestimine tähendab sisuliselt seda, et ideaalis testitakse läbi kogu tarkvara kood. Selleks luuakse eraldi testide süsteem, kuhu kirjutatakse automaatteste, mis võiks katta ära võimalikult suure osa tarkvara funktsionaalsusest.

Töö autor töötab organisatsioonis, kus on kasutusel mahukas monoliitse arhitektuuriga pangandustarkvara. Tarkvaras esineb palju vigu, mille üheks põhjuseks on olemasoleva automaattestimissüsteemi ebaefektiivsus, kuna süsteem ei kata piisavalt palju koodi ning vähe on kaetud ka erinevaid teststsenaariumeid.

Diplomitöös käsitletakse regressioonitestimist just seepärast, et organisatsioonis on tavaks saanud iganädalane uue tarkvaraversiooni väljastus. Seetõttu on kehtestatud nõue kasutada uue versiooni kontrollimisel regressiooniteste, et veenduda selle kvaliteedis.

Diplomitöö eesmärk on töötada välja efektiivsem rakendusliideste regressioontestimise tarbeks mõeldud testimissüsteem autori tööandja näitel. Töö eeskujul on võimalik ka teistel inimestel ja organisatsioonidel koostada tõhus automaattestimissüsteem, et rakenduse kõrge kvaliteet

oleks kontrollitud ja igal ajal oleks võimalik veenduda tarkvara toimimises ning uute arenduste võimalike vigade vältimises.

Autor püüab uurimistööga leida vastused järgnevatele uurimisküsimustele:

- Millised eelised on regressioonitestidel rakendusliidestese testimisel võrreldes teiste testimismetoodikatega?
- Milliste vahenditega saab muuta regressiooniteste võimalikult efektiivseks?
- Milliseid tööriistu tuleks eelistada automaattestide loomisel?

Uurimisküsimustest tulenevalt on püstitatud järgnevad uurimisülesanded:

- võrrelda erinevaid testimisraamistikke, tööriistu ja metoodikaid, viia läbi katsetused, teha analüüs ning järeldused;
- selgitada välja vahendid ja meetodid regressioonitestide efektiivsuse tagamiseks;
- koguda materjali regressioonitestide efektiivsemaks muutmise kohta ning viia ise läbi katsetused, neid analüüsida ning teha järeldused.

Uurimistöös käsitletakse tarkvara testimist üldiselt ning põhjalikumalt just rakendusliideste automaattestimist. Töö autor võrdleb erinevaid automaattestimise metoodikaid ning selleks kasutatavaid tööriistu, protsesse ning testimisraamistikke. Autor täiustab oma töökohas kasutusel olevat rakendusliideste testimissüsteemi uute testidega ning lisab koodinäiteid. Analüüsib tehtut ning põhjendab näidetega, kuidas loodud arendus süsteemi täiustab ning mille tulemusel on tarkvara veakindlam ja toimekam.

1. TARKVARA TESTIMISE METOODIKAD

Tarkvara testimine on protsess, mille käigus hinnatakse tarkvara kvaliteeti ja toimivust, tuvastatakse vigu, kontrollitakse nõuete täitmist ning tagatakse tarkvara vastavus konkreetsetele standarditele ja spetsifikatsioonidele. Selle käigus kasutatakse erinevaid testimismeetodeid ja tööriistu, et tagada tarkvara toimivus, usaldusväärsus ja kasutuskogemus enne selle lõplikku väljatoomist või kasutuselevõttu (M. Pezzè, M. Young, 2008).

Tarkvara testimise protsess hõlmab mitmeid meetodeid ja strateegiaid, mida kasutatakse selleks, et tagada rakenduse vastavus kliendi ootustele. Testimismeetodite hulka kuuluvad funktsionaalne ja mittefunktsionaalne testimine, valideerimaks rakenduse töökorras olekut. Kasutusel on mitmeid testimise meetodeid, nagu näiteks ühiktestimine, integratsioonitestimine, süsteemi testimine, jõudluse testimine, regressioonitestimine jne. Igal testimismeetodil on paikapandud eesmärk, strateegia ja soovitud tulemused (T. Hamilton, 2023).

1.1. Tarkvaraarenduse mudelid ja testimispraktikad

Järgnevalt tuuakse välja mõned üldkasutatavad tarkvaraarenduse mudelid ning nende jaoks optimaalseimad testimispraktikad.

Agiilne metoodika

Traditsioonilised tarkvaraarenduse metoodikad, nagu näiteks koskmudel (tarkvaraarenduse protsessi mudel, mis kirjeldab arenduse etappe lineaarse jada kaudu, mis liiguvad üksteise järel nagu kosk), lähtuvad eeldusest, et tarkvara nõuded jäävad projekti jooksul muutumatuks. Koos keerukuse suurenemisega võivad nõuded sageli muutuda ja pidevalt areneda. Mõnikord ei ole isegi klient kindel, mida ta lõpuks soovib. Ehkki iteratiivne mudel seda probleemi käsitleb, põhineb see metoodika siiski koskmudelil. Agiilses metoodikas arendatakse tarkvara järkjärguliselt ja kiiretes tsüklites. Tarkvaraarenduse tsüklit nimetatakse ka *sprindiks*, see on lühike, ajaliselt piiritletud tarkvaraarendusperiood, mille jooksul arendustiim töötab konkreetsete eelnevalt kokkulepitud ülesannetega. Rõhku pannakse suhtlusele klientide, arendajate ja tellijate vahel, mitte niivõrd protsessidele ja tööriistadele. Agiilne metoodika keskendub muutustele reageerimisele, mitte ulatuslikule planeerimisele (T. Hamilton, 2023).

Agiilses arendusmeetodis on kasutusel inkrementaalne testimine ja seetõttu testitakse igat projekti väljalaset põhjalikult. See tagab, et võimalikud vead parandatakse enne järgmist väljalaset. Eeliseks on võimalus teha projektis muudatusi igal ajal vastavalt nõuetele. Puuduseks on pidev tellijaga suhtlemine, mis tähendab lisa ajakulu kõigile osapooltele, sealhulgas kliendile endale (T. Hamilton, 2023).

Ekstreemprogrammeerimine

Ekstreemprogrammeerimine on üks agiilse metoodika alaliike, mis toetab lühikesi arengutsükleid. Projektid jagatakse lihtsateks ülesanneteks. Arendajad kirjutavad lihtsa jupi programmi ja saadavad selle kliendile tagasiside saamiseks. Kliendi tagasiside põhjal tehakse parandused ja jätkatakse järgmise ülesandega. Ekstreemprogrammeerimise puhul töötavad arendajad tavaliselt paarides. Seda kasutatakse siis, kui kliendi nõuded muutuvad pidevalt (T. Hamilton, 2023).

Arenduse käigus järgitakse testipõhist arendust, mis tähendab, et kõigepealt kirjutatakse valmis ühiktestid ja seejärel rakenduskood, et testitingimusi täita. Eeliseks on pidev testimine ja pidev versioonide väljalase, mis tagavad koodi kõrge kvaliteedi. Sobib olukorras, kus tellijal on ebamäärased soovid. Puuduseks on tihedatele tellijaga kohtumistele kuluv aeg (T. Hamilton, 2023).

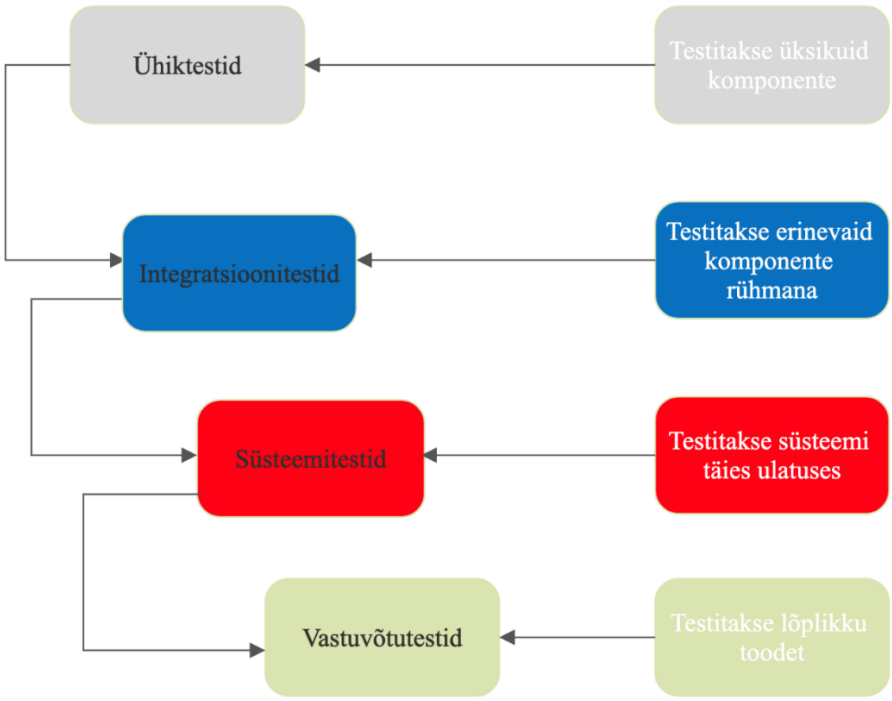
1.2. Testimise tasemed

Testimise tasemed määratlevad testimise ulatust ehk kui suurt osa rakendusest testitakse. Iga tase keskendub konkreetsetele testimise aspektidele. Tasemeid järjestatakse vastavalt testimise ulatusele ning nendeks peetakse:

1. ühiktestimist;
2. integratsioonitestimist;
3. süsteemitestimist;
4. vastuvõtutestimist.

Need tasemed hõlmavad funktsionaalset ja mittefunktsionaalset testimist, manuaalset ja automatiseeritud testimist, erinevaid testimisstrateegiaid ja -meetodeid nagu näiteks valget ja

musta kasti testimist ning palju muud. Igal testimise liigil on oma eesmärgid, eelised ja puudused ning sobiv valik sõltub tarkvara arendusprotsessist ja klientide vajadustest (M. A. Umar, 2019).



Joonis 1 Testimise tasemed vooskeemina (International Software Test Institute, 2023).

1.2.1. Ühiktestid

Ühiktestimine on tarkvara testimise kõige madalam tase, kus testitakse üksikuid tarkvarakomponente iseseisvalt ehk eraldi kogu rakendusest. Selle eesmärk on kinnitada, et iga tarkvara osa toimib nii nagu vaja. Ühiktestimist tehakse enne integratsiooniteste. Kuigi ühiktestid luuakse üldjuhul pärast funktsionaalse koodi kirjutamist, siis testipõhise arenduse korral kirjutatakse testid enne rakenduskoodi. Tavaliselt loovad ühiktestid tarkvaraarendajad. Harvematel juhtudel võib seda teha ka meeskonnas määratud testija (Software Testing Fundamentals, 2023).

Ühiktestimisel on mitmeid olulisi tegevusi (Software Testing Fundamentals, 2023):

- suurendab kindlustunnet koodi muutmise ja haldamise osas. Kui ühiktestid on head ja need käivitatakse iga kord kui koodi muudetakse, siis on võimalik kiiresti leida defekte, mis võivad olla sisse viidud muudatuste tõttu. Kui koodi komponendid on omavahel vähem seotud, siis on ühiktestide loomine lihtsam ning on väiksem tõenäosus teha koodi tahtmatuid muudatusi;

- kood muutub rohkem taaskasutatavaks, kuna ühiktestide jaoks tuleks koodi moduleerida (jaotamine väikesteks ja teineteisest võimalikult vähe sõltuvateks osadeks), mis omakorda lihtsustab koodi taaskasutamist;

- arendusprotsess võib kiireneda. Kuigi testide kirjutamine on ajamahukas, siis kokkuvõttes võtab protsess vähem aega, kuna arendajad ei pea enam kasutajaliidese kaudu uut funktsionaalsust erinevate sisenditega käsitsi katsetama. Pikas perspektiivis on arendus kiirem, kuna ühiktestimise käigus leitud vigade parandus nõuab vähem pingutust võrreldes süsteemi- või vastuvõtutestidest leitud vigadega;

- silumine on lihtsam, kuna ühiktesti ebaõnnestumise korral tuleb silumisel vaadata vaid viimaseid muudatusi. Kui viga avastatakse testimisprotsessi hilisemas etapis, on silumine sellevõrra keerulisem.

1.2.2. Integratsioonitestid

Integratsioonitestimine on testimise tase, kus individuaalsed üksused/komponendid ühendatakse ja testitakse rühmana. Selle taseme eesmärk on avastada vead, mis võivad tekkida integreeritud komponentide omavahelises koostöös. Testimise abistamiseks kasutatakse testijuhte (inglise keeles *test drivers*) ja simulatsiooniprogramme (inglise keeles *test stubs*) (M. A. Umar, 2019).

Integratsioonitestid viiakse läbi pärast ühikteste ning tavaliselt on see protsess automatiseeritud. Iteratiivse arendustsükli korral on integratsioonitestid osa pideva integratsiooni protsessist (M. A. Umar, 2019).

1.2.3. Süsteemitestid

Süsteemitestimine on testimise tase, kus testitakse tarkvara täies ulatuses läbi veendumaks, et kogu süsteem töötab ettenähtud viisil ja vastab määratud nõuetele. Süsteemitestid viiakse üldiselt läbi testijate poolt ning see on kõige põhjalikum testimise tase, mille raames kasutatakse erinevaid testimise tüüpe, nagu näiteks (Software Testing Fundamentals, 2023):

- proovitestimine (inglise keeles *smoke testing*) - hindab ja testib kõige olulisemaid funktsioone;
- funktsionaalne testimine - eesmärk on tagada, et funktsionaalsused töötavad vastavalt nõuetele;
- regressioonitestimine - eesmärk on veenduda, et uued koodimuudatused ei kahjusta juba olemasolevat arendust;
- kasutatavuse testimine (inglise keeles *usability testing*) - hindab kasutajaliidese mugavust lõppkasutaja vaatest;
- jõudlustestimine - eesmärk on hinnata süsteemi reageerimisvõimet ja stabiilsust erinevat laadi koormuste all;
- turbetestimine - eesmärk on avastada süsteemi haavatavused ja tagada selle kaitse pahatahtlike rünnakute eest;
- vastavustestimine (inglise keeles *compliance testing*) - eesmärk on kontrollida, kas lahendus vastab konkreetsetele nõuetele, standarditele ja regulatsioonidele.

Kuna süsteemitestimine hõlmab tarkvara testimist tervikuna, peaks testkeskkond olema võimalikult sarnane tootmiskeskkonnaga (inglise keeles *production environment*).

1.2.4. Vastuvõtutestid

Vastuvõtutestid viiakse üldjuhul läbi äriosakonna inimese poolt ning selle eesmärgiks on kindlaks teha, kas loodud süsteem vastab vastuvõetavuse kriteeriumidele (inglise keeles *acceptance criteria*) ehk kindlaks määratud tingimustele, millele toode peab vastama (Scrum

Alliance, 2023). Need testid võimaldavad kasutajal otsustada, kas arendus tuleks vastu võtta või mitte. Vastuvõtutestid on kogu testimise protsessis viimane samm ja need tehakse alati manuaalselt (M. A. Umar, 2019).

2. REGRESSIOONITESTIMINE

Regressioon tarkvaraarenduses viitab olukorrale, kus pärast tarkvara arendamist või muudatuste tegemist tekivad uued vead või kahjustuvad varasemalt arendatud funktsionaalsused. Regressioonitestimine sisaldab korduvate testide jooksutamist ning selle eesmärk on kontrollida, kas varem toiminud tarkvara töötab endiselt pärast uusi koodimuudatusi. IEEE (Elektri- ja Elektroonikainseneride Instituut) standardi kohaselt on regresioonitestimine selektiivne süsteemi või komponendi testimine kontrollimaks, et muudatused pole põhjustanud soovimatuid tagajärgi ning süsteem või komponendid vastavad endiselt määratletud nõuetele. Fookus võib olla kas testjuhtumite uuesti käivitamisel või funktsionaalsuse uuesti testimisel. Üldiselt võib regressioonitesti eesmärk erineda erinevate organisatsioonide või organisatsiooni osade vahel. Eesmärk võib olla kas defektide avastamine või süsteemi kvaliteedi mõõtmine (E. Engström, 2010).

Efektiivne regressioonitestimine on oluline ja isegi kriitiline neile organisatsioonidele, kes tegelevad peamiselt tarkvaraarendusega. See sisaldab muuhulgas vajalike testjuhtude määramist, st. regressioonitesti valikut, et kontrollida muudetud tarkvara käitumist. Iteratiivne arendustsükkel ja koodi taaskasutus on levinud viisid aja ja vaeva säästmiseks tarkvaraarenduses, kuid mõlemal juhul on nõutud sagedane eelnevalt testitud funktsioonide uuesti testimine koodi muudatuste tõttu. Seetõttu muutub tõhusa regressioonitestimise vajadus aina olulisemaks (E. Engström, 2010).

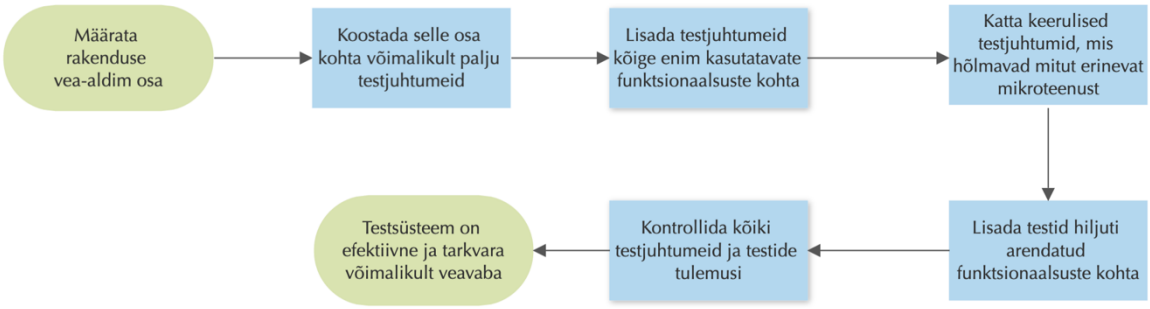
2.1. Regressioonitestimise protsess

Regressioonitestide jooksutamiseks on vajalik luua testsüsteem ehk kogum testjuhtumeid, mis on automatiseeritud ja mida saab erinevate tööriistade abil testimiseks käivitada. Sisuliselt võib testsüsteem olla eraldiseisev projekt mõnes integreeritud arenduskeskkonnas (inglise keeles *Integrated Development Environment, IDE*), vastavalt organisatsiooni nõuetele, kuhu luuakse testjuhtumeid, mis näiteks rakendusliideste testimise korral on seadistatud vastu tarkvara rakendusliideste lõpp-punkte (inglise keeles *API endpoint*). Testjuhtum sisaldab funktsionaalset koodi, mis antud näite korral testib rakendusliideste lõpp-punkte erinevate parameetritega. Üks lihtne näide (koodinäide 1) taolisest testjuhtumist on järgnevas koodinäites:

```
@Test
public void somethingIsCreatedAndRejected(ITestContext context) {
    SomePayloadDto customer = (SomePayloadDto) context.getAttribute(CUSTOMER.name());
    Long versionId = someService.createSomething(customer);
    ByVersionIdResponse byVersionIdResponse = someApi.getSomethingByVersionId(versionId
        );
    someApi.deleteSomeOpeningProcess(byVersionIdResponse.getId(),
        "REJECTED",
        "SIGNING_FAILED");
    byVersionIdResponse = someApi.getSomethingByVersionId(versionId);
    Assert.assertEquals(byVersionIdResponse.getStatusCode(), "REJECTED");
}
```

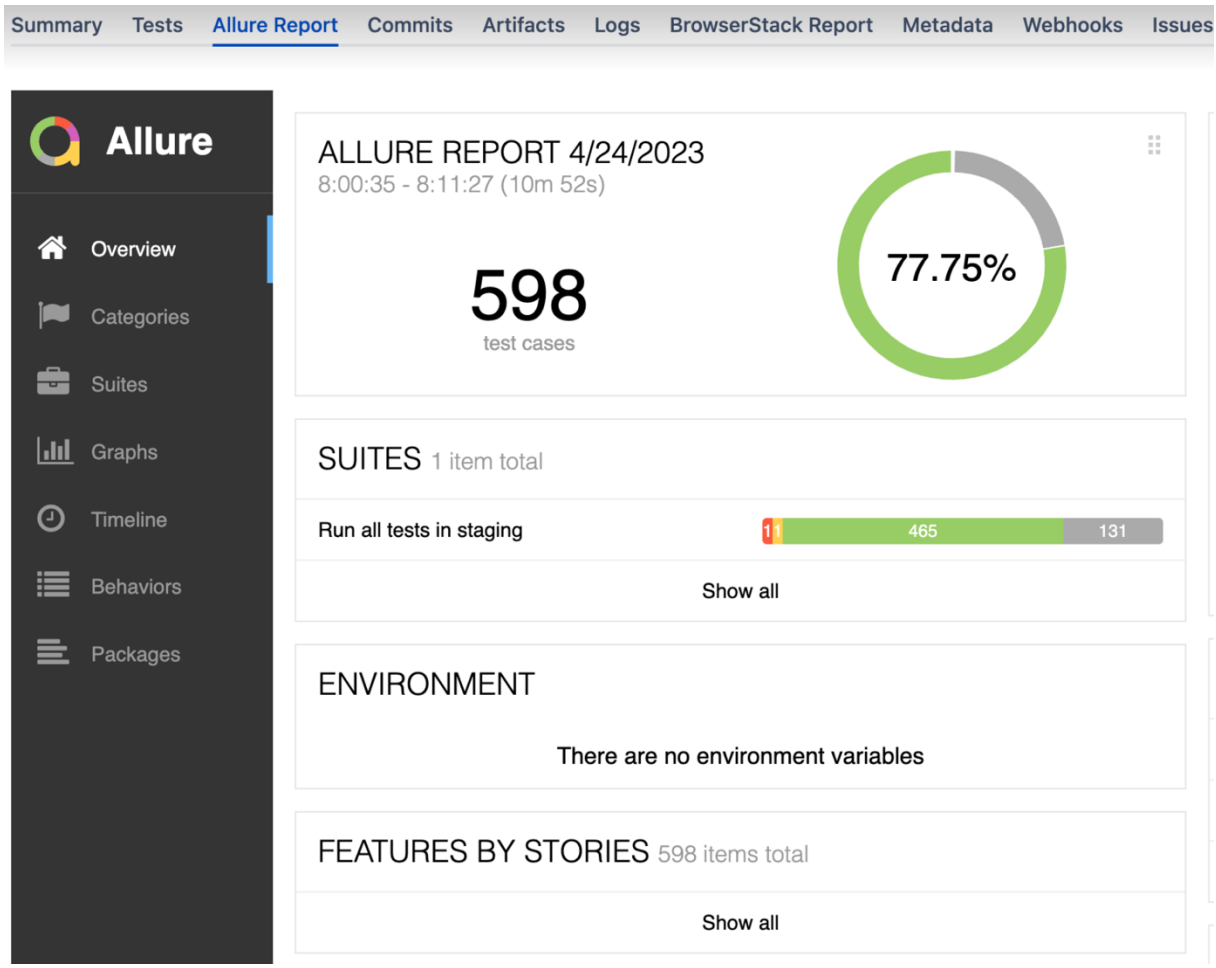
Koodinäide 1 Näide rakendusliidese testjuhtumist.

Testjuhtumite koostamisel tuleks esmalt välja selgitada, milline osa rakendusest on kõige vea-altim ning koostada selle kohta võimalikult palju juhtumeid. Järgnevalt tuleks lisada juhtumid tarkvaras kõige enim kasutatavate funktsionaalsuste kohta. Ära tuleks katta ka keerulised testjuhtumid (näiteks sellised rakenduse funktsionaalsused, kus on kasutusel mitu erinevat mikroteenust). Hea oleks kirjutada testid ka hiljuti arendatud funktsionaalsuste kohta, et tagada jätkuvalt põhieesmärk - regressiooni vältimine.



Joonis 2 Testjuhtumite koostamise vooskeem (Software Testing Help, 2023).

Testsüsteemi tuleks regulaarselt uuendada, näiteks siis, kui tarkvarast leitakse uus defekt, mida olemasolevat testjuhtumid ei kata. Sellisel juhul on kindel, et vähemalt see osa koodist saab kaetud ja järgmisel korral sama viga ei teki. Teste tuleks jooksutada alati siis, kui koodis on tehtud muudatusi, defekt on parandatud, lisatud on uusi funktsionaalsusi või olemasolevat funktsionaalsust on täiendatud. Hea oleks koostada ka testitulemuste aruanne, mis sisaldab läbiviidud testjuhtumite edukust ja ebaõnnestumist (Software Testing Help, 2023).



Joonis 3 Testitulemuste aruanne, kus on kuvatud testjuhtumite arv ja edukuse protsent.

2.2. Regressioonitestimise eelised

Regressioonitestimine mängib olulist rolli agiilses arendustsüklis, kus igas *sprindis* tuleb tagada integreerumine eelneva ja uue versiooni vahel. Järgnevalt on välja toodud regressioonitestide eelised (Testsigma, 2023):

- regressioonitestidega muudetakse tarkvara vigade suhtes vastupidavamaks, mis toetab testijaid toote uue versiooni kiirema ja kvaliteetsema väljastamisega;
- tagatud on rakenduse terviklikkuse ja integreerituse säilimine pidevate täienduste korral. Automaatsed testid genereerivad kiireid tulemusi, mis aitab jälgida pidevalt uute vigade teket. Lisaks on testimisprotsess oluliselt suurenenud ja kogu testimistsükkel on lühem;

- automaattestid võivad joosta öösiti ja mitmes masinas korraga samal ajal, seega ei pea testijad sellele enam liialt tähelepanu pöörama ning saavad keskenduda teistele tarkvara kitsaskohtadele;
- aitab vähendada ebavajalikke kulusid, mis on seotud tootmiskeskkonnas juhtunud intsidentidega. Automaatse tarkvara testimise lahenduste kasutamine vähendab ka projekti kogukulu;
- aitab tõsta tarkvara usaldusväärsust ja stabiilsust. Kui uute funktsioonide lisamisel tekivad vead, siis regressioonitestimine aitab need kiiresti tuvastada ning kõrvaldada. See omakorda aitab tarkvara kasutajatele tagada parema kasutuskogemuse ning vähendab negatiivset mõju äritegevusele;
- aitab paremini mõista tarkvara erinevate komponentide toimimist, mis aitab arendajatel tulevikus efektiivsemalt töötada ning tarkvara paremini hooldada.

Kokkuvõttes on regressioonitestidel mitmeid eeliseid, sealhulgas tarkvara vastupidavuse suurendamine vigade suhtes ning terviklikkuse ja integreerituse säilitamine pidevate täienduste korral. Automaatsed testid kiirendavad testimisprotsessi, võimaldades testijatel keskenduda muudele tarkvara probleemidele. Regressioonitestimine vähendab kulusid, mis on seotud tootmiskeskkonnas esinevate intsidentidega, suurendades samal ajal tarkvara usaldusväärsust ja stabiilsust. Lisaks aitab see arendajatel paremini mõista tarkvara erinevate komponentide toimimist, võimaldades tulevikus efektiivsemat tööd.

3. RAKENDUSLIIDES JA SELLE TESTIMINE

Rakendusliides (inglise keeles *Application Programming Interface*, lühendatult API) on tarkvara komponent, mis võimaldab erinevatel rakendustel ja süsteemidel omavahel suhelda ja andmeid vahetada. Rakendusliidesed võivad olla erinevat tüüpi nagu RESTful, SOAP, GraphQL jne (J. Varun Iyer, 2023).

Rakendusliides annab võimaluse tarkvaraarendajatel luua rakendusi, mis kasutavad teiste rakenduste ja süsteemide funktsionaalsust. Näiteks võib veebirakendus kasutada Google Maps API-t, et kuvada kaardiandmeid oma veebilehel või mobiilirakendus kasutada Facebook API-t, et võimaldada kasutajatel sisse logida oma Facebook´i konto kaudu. Rakendusliideste abil saavad ka ühe süsteemi erinevad teenused omavahel suhelda ja koostööd teha. Näiteks võib üks teenus kasutada teise teenuse API-t, et saada juurdepääs selle andmetele või funktsionaalsusele. See aitab teenustel töötada koos tervikliku lahendusena, ilma et tuleks eraldi luua igat tarkvara osa (A. Walker, 2023).

3.1. Rakendusliideste protokollid

Rakendusliidestel on kindlaks määratud kutsungid (inglise keeles *API call*) ja meetodid, mida arendajad kasutavad rakendusliideste edukaks integreerimiseks oma programmidesse. Reeglid, mis juhivad nende kutsungite kasutamist ja rakendamist nimetatakse rakendusliideste protokollideks. Protokollid määratlevad lubatud käsud ja andmetüübid ja seavad kindla standardi rakendusliidese kasutamiseks (J. Varun Iyer, 2023).

Erinevad üldkasutatavad rakendusliideste protokollid on:

- SOAP;
- REST;
- GraphQL.

Järgnevalt igast protokollist pisut lähemalt.

3.1.1. SOAP

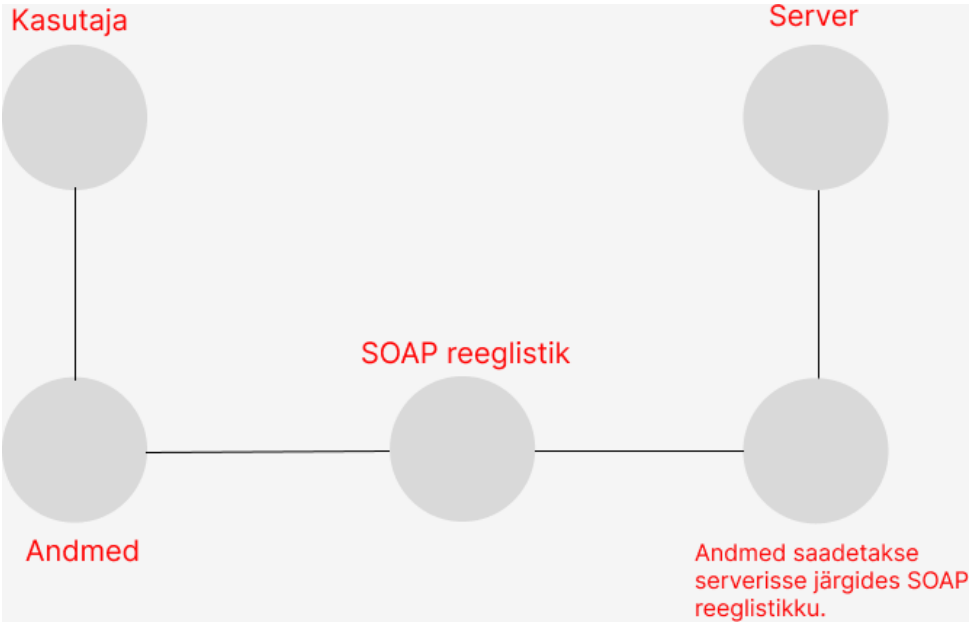
SOAP (*Simple Objects Access Protocol*) on protokoll, mis kasutab suhtluseks XML-i. See on kõige vanem rakendusliideste protokoll, mis loodi 1998. aastal. SOAP kasutab XML-faile andmete edastamiseks veebiteenuste vahel. Need XML-failid jagatakse laiali üle HTTP/HTTPS ühenduste. Siiski võimaldab SOAP ka andmete edastamist teiste protokollidega nagu Transmission Control Protocol (*TCP*), Simple Mail Transport Protocol (*SMTP*), User Data Protocol (*UDP*) jne. SOAP protokolli kasutavad rakendusliideseid nimetatakse SOAP API-deks (J. Varun Iyer, 2023).

SOAP sõnumid on kodeeritud XML-is ja neil on kindel formaat (J. Varun Iyer, 2023):

- ümbris: sisaldab sõnumi andmeid ja markeerib selle alguse ja lõpu;
- päis: määratleb kogu lisateabe, mida võib andmetöötluseks vaja minna. See on valikuline;
- keha: kirjutatakse päring vajalike andmete kättesaamiseks;
- viga: määratleb kõik vead, mis võivad andmete edastamise ajal tekkida, ja meetmed nende käsitlemiseks.

```
HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>

<soap:Envelope
xmlns:soap="http://www.w3.org/2003/05/soap-envelope/"
soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">

<soap:Body xmlns:m="http://www.example.org/stock">
  <m:GetStockPriceResponse>
    <m:Price>34.5</m:Price>
  </m:GetStockPriceResponse>
</soap:Body>

</soap:Envelope>
```

Koodinäide 2 Näide SOAP sõnumist (W3Schools, 2003).



Joonis 4 SOAP visualisatsioon (J. Varun Iyer, 2023).

Kuigi SOAP on veebis väga levinud ja see võimaldab andmete edastamist mitmete protokollidega, muudab selle sõltuvus XML-ist kasutamise väga piiratuks, kuna XML-il on range vorming. Raskeks võib osutuda ka XML-i tõrgete silumisega seotud takistuste lahendamine (J. Varun Iyer, 2023).

3.1.2. REST

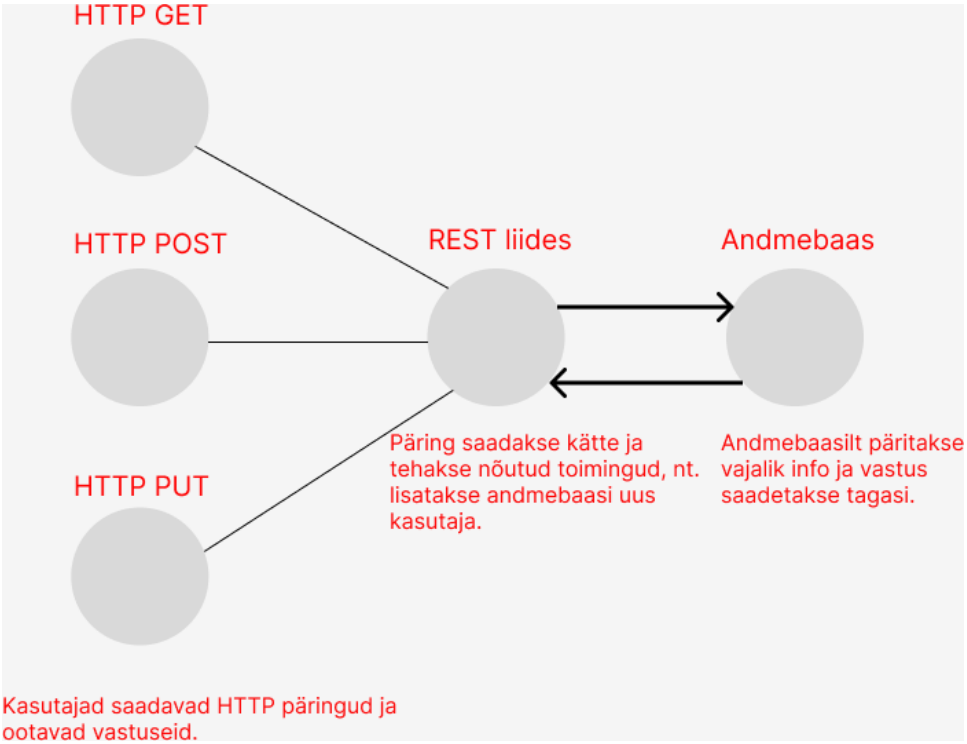
REST (Representational State Transfer) on arhitektuuristiil, mida kasutatakse veebirakenduste arendamisel. REST protokollid lahendavad SOAP protokolli XML-st sõltuvuse probleemi, toetades andmete edastamist mitmesugustes formaatides nagu JSON (kõige populaarsem), HTML, lihttekst ja ka meediafailid. REST toetub aga ainult HTTP/HTTPS andmeedastusele, võttes ära SOAP-i kohanemisvõime teiste protokollidega. REST-i protokolli kasutavad rakendusliideseid nimetatakse RESTful API-deks (J. Varun Iyer, 2023).

REST API-d järgivad klient-server arhitektuuri ja peavad olema olekuvabad. Olekuvaba suhtlus tähendab, et GET päringute vahel ei salvestata kliendiandmeid. Need GET päringud peavad olema eraldiseisvad ja eristatavad. REST määrab igale toimingule unikaalse URL-i, nii et kui server saab päringu, teab see, millised juhised tuleb täita. REST toetab ka vahemälu kasutamist.

Seega saab brauser tulemused päringust salvestada kohalikult ja neid perioodiliselt vastavalt vajadusele taastada, suurendades seeläbi kiirust ja tõhusust (J. Varun Iyer, 2023).

Tüüpilisel REST-päringul on järgmised komponendid (J. Varun Iyer, 2023):

- lõpp-punkt: URL, kust andmeid küsitakse;
- meetod: kasutatakse eelnevalt määratletud meetodeid, nagu GET, POST, PUT või DELETE andmete hankimiseks. Need meetodid erinevad üksteisest. Näiteks GET-i kasutamisel lisatakse andmed URL-i lõppu, samas kui POST-i korral saadetakse andmed koos HTTP-päringuga;
- päised: määratlevad päringu üksikasjad ja dikteerivad vastuse õige formaadi;
- keha (andmed): tegelikud andmed, mida teenus saadab.

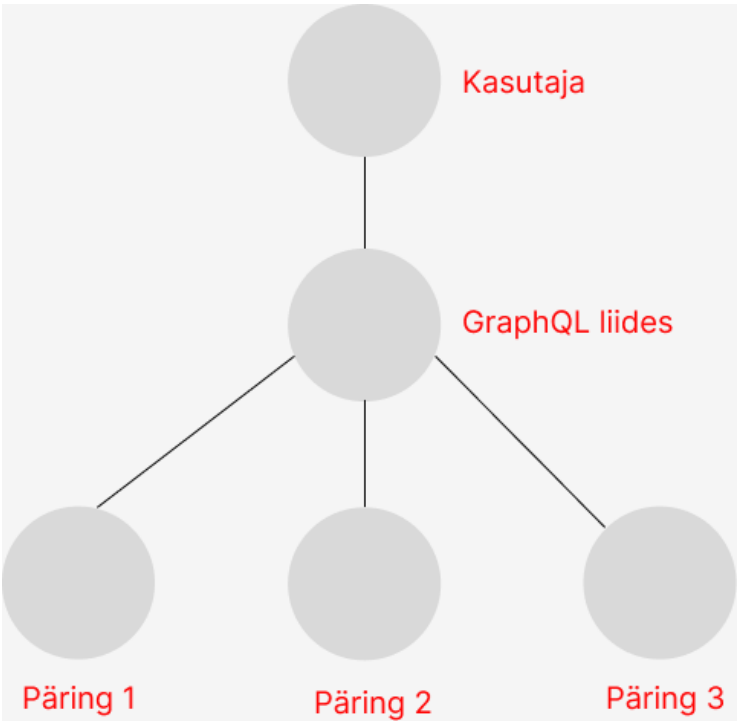


Joonis 5 REST liidese visualisatsioon (J. Varun Iyer, 2023).

REST API-d pakuvad rakenduste arendajatele rohkem vabadust, on kerged ja skaleeritavad (J. Varun Iyer, 2023).

3.1.3. GraphQL

GraphQL ehk *Graph Query Language* on Facebooki poolt välja töötatud ning avaldatud 2015. aastal. See võtab huvitava lähenemise rakendusliideste kommunikatsiooni võimaldamiseks. Sarnaselt andmebaasi päringukeeltele nagu SQL küsib GraphQL andmeid serverilt. Kasutaja määratleb päringus vajalikud andmed ning formaadi ja GraphQL tagastab andmed samas formaadis nagu päringus nõutud (J. Varun Iyer, 2023).



Joonis 6 GraphQL visualisatsioon (J. Varun Iyer, 2023).

See säästab aega ja mälu, kuna serverist küsitakse ainult neid konkreetseid andmed, mis olid päringus nõutud, mitte mahukaid eelpakendatud faile koos ebavajaliku lisainformatsiooniga. GraphQL on välja töötatud toetamaks erinevaid veebiarenduskeeli. Probleeme võib tekkida andmete vahemällu salvestamisega. Kuna andmeid küsitakse väga konkreetsete päringutega, võib tekkida vajadus küsida teisi seotud andmeid eraldi, kuna neid varasemalt ei päritud ning seega ei leita neid ka vahemälust (J. Varun Iyer, 2023).

3.2. Rakendusliideste automaattestimine

Rakendusliideste manuaalne testimine on väga ajakulukas protsess. Kuna käsitsi testimine tehakse inimese poolt, on suur oht vigade ja eksimuste tegemiseks. Suure hulga andmete käsitsi võrdlemine on võimatu. Käsitsi rakendusliideste testimine pole sobiv suurtele organisatsioonidele ja ajaliselt piiratud projektidele. Isegi üks valesti töötav rakendusliides võib mõjutada kogu toodet. Seetõttu on käsitsi rakendusliideste testimine aegunud ja eelistatav on automatiseeritud rakendusliideste testimine. Automatiseeritud rakendusliideste testimine on mitte ainult kiire protsess, vaid vähendab ka vigade tõenäosust. Automatiseeritud rakendusliideste testimine ei vaja testija füüsilist kohalolu ja seda saab teha igal ajal ja igas kohas (Isha, A. Sharma, M. Revathi, 2018).

Viimaste aastate tarkvaraarenduse trend on olnud muuta rakendused hajusaks, kasutades standardiseeritud liideseid (nt REST), et suhelda teiste tarkvarakomponentidega, näiteks erinevate teenuste või andmebaasidega, mis asuvad erinevates serverites. Tarkvara loojad pakuvad oma andmeid ja äriloogikat kasutades rakendusliideseid, et teenida lisatulu. Seda nähtust nimetatakse API majanduseks (inglise keeles *API economy*). Rakendusliideste arv kasvab aasta-aastalt kiiresti ning seetõttu on automatiseeritud rakendusliideste testimine muutunud oluliseks tarkvaraarenduse osaks. Valesti toimivad või ebaefektiivsed rakendusliidesed võivad vähendada toodete müüki ja tulusid (Isha, A. Sharma, M. Revathi, 2018).

Rakendusliideste testimine on oluline, et tagada äriloogika funktsionaalsus, usaldusväärsus, turvalisus ja edastus. Peamised väljakutsed, millega automatiseeritud rakendusliideste testimisel silmitsi seistakse (Isha, A. Sharma, M. Revathi, 2018):

- rakendusliidese kutsungite järjestamine - näiteks testides e-poe rakendusliidest, siis tuleb hoolikalt järjestada kõik ostukorvi lisamise ja makseprotsessi kutsungid, et tagada edukas tellimus;
- ettearvamatute vastuste võrdlemine - rakendusliides võib tagastada erinevaid vastuseid sõltuvalt sisendist, näiteks testides panga rakendusliidest võivad vastused erineda sõltuvalt kasutaja kontol olevast summast või tehingu tüübist;

- etteplaneeritud nurjunud testjuhtumite käsitlemine - testides tarkvara, mis peab tagama andmete privaatsuse, tuleb ette planeerida kuidas rakendusliides käitub, kui sellist tüüpi testjuhtumid nurjuvad;

- rakendusliideste paralleelne käivitamine - kui testitakse suurt andmekogumit, mis nõuab suurt hulka kutsungeid, peaks aja kokkuhoiu mõttes neid paralleelselt käivitama;

- parameetrite sõltuvuse käsitlemine - ühe liidese kutsungi vastus võib sõltuda teise liidese kutsungi eelnevast sisendist. Näiteks kui testitakse kasutaja autentimist enne mingi tegevuse sooritamist, siis tuleks tagada, et kõik kutsungid käituvad korrektselt ja parameetrid oleks õigesti seadistatud.

4. AUTOMAATTESTIDE KIRJUTAMINE RAKENDUSLIIDESTELE

Rakendusliideste automaattestide loomine on keeruline ning ajamahukas protsess ja selleks on saadaval mitmeid erinevaid tööriistu ja testraamistikke. Need tööriistad ja raamistikud võimaldavad arendajatel ja testijatel kirjutada ning käivitada automaatseid teste vastu rakenduse lõpp-punkte, tagades nende õige toimimise ja oodatud nõuete täitmise. Töö autor vajab rakendusliideste regressioonitestide automatiseerimiseks tööriistu ja peab valima sobivaimad. Järgnevas peatükis tuuakse välja rakendusliideste automaattestide kirjutamiseks erinevad tööriistad ja raamistikud.

4.1. Testimistööriistade võrdlus ja valik

Testimise tööriistad on vahendid, mida kasutatakse testimisprotsessi automatiseerimiseks, andmete haldamiseks ning mis abistavad testide analüüsiga (D. Atesogullari, A. Mishra, 2020). Tööriistade valikul on arvestatud mitmete teguritega, sealhulgas nende populaarsusega ja erinevatele tingimustele vastamine, nende seas lihtne kasutatavus, hea dokumentatsioon ja sobivus Java keelega. Populaarsus on oluline tegur, kuna see võib näidata tööriista usaldusväärsust ja aktiivset kasutajate kogukonda. Samuti võib populaarsete tööriistade puhul olla lihtsam leida ressursse, õpetusi ja dokumentatsiooni, mis aitavad nende kasutamist õppida ja tõhusamalt kasutada.

Internetist leitavate testijate kogukonna materjalide alusel võib populaarsemateks testimistööriistadeks pidada (Software Testing Material, 2023):

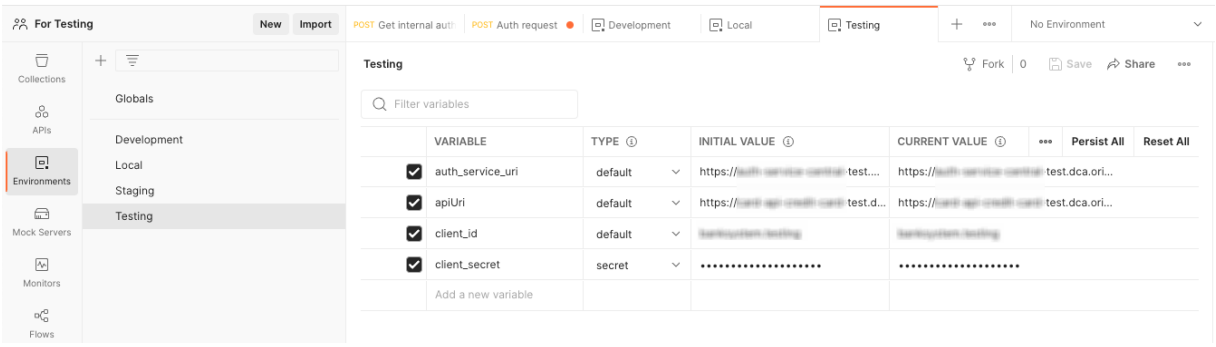
- Postman;
- REST-assured;
- JMeter;
- SOAP UI.

4.1.1. Postman

Üks populaarsemaid rakendusliideste testimise tööriistu on Postman. See 2012. aastal loodud programm võimaldab kasutajatel hõlpsasti töötada kõikide rakendusliidestega seotud osadega

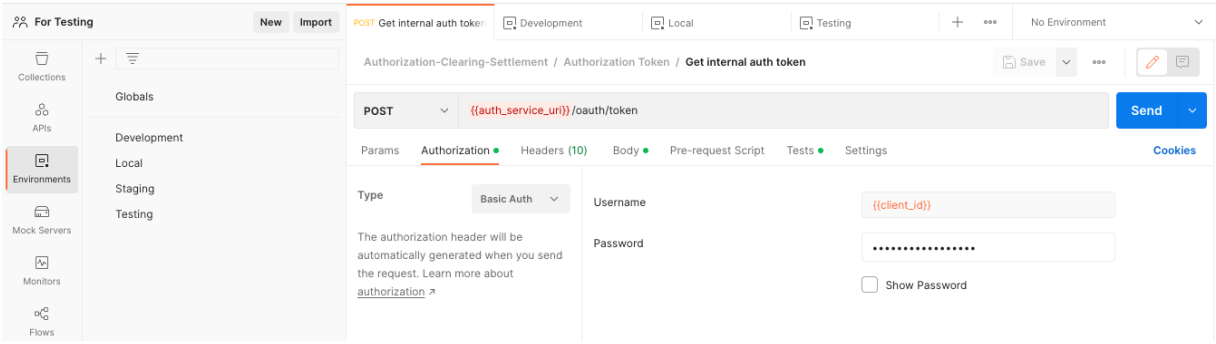
(nt dokumentatsioon, testjuhtumid, tulemused) ühel kesksel platvormil. Postmanil on kasutajasõbralik kasutajaliides, kus rakendusliidese elutsükli iga olulise osa visualiseerimine ja mõistmine on tehtud võimalikult lihtsaks. Postman pakub testimiseks laia valikuvõimalust - testida saab erinevates keskkondades ja serverites, testidele on võimalik luua dokumentatsiooni ning kergeks on tehtud ka kõige vajaliku jagamine meeskonnakaaslaste vahel (D. Sekulovski, 2023).

Töö autor on seadnud üles erinevad keskkonnad rakendusliideste testimiseks - testkeskkond, arenduskeskkond, proovikeskkond (inglise keeles *staging environment*) ja kohalik keskkond (juhul, kui rakendus on käivitatud kohalikust masinast). Määratud on muutujad ja nendele vastavad väärtused, näiteks keskkonnale vastav aadress või kasutajakonto nimi ja parool.



Joonis 7 Postman keskkonna seadistuste vaade.

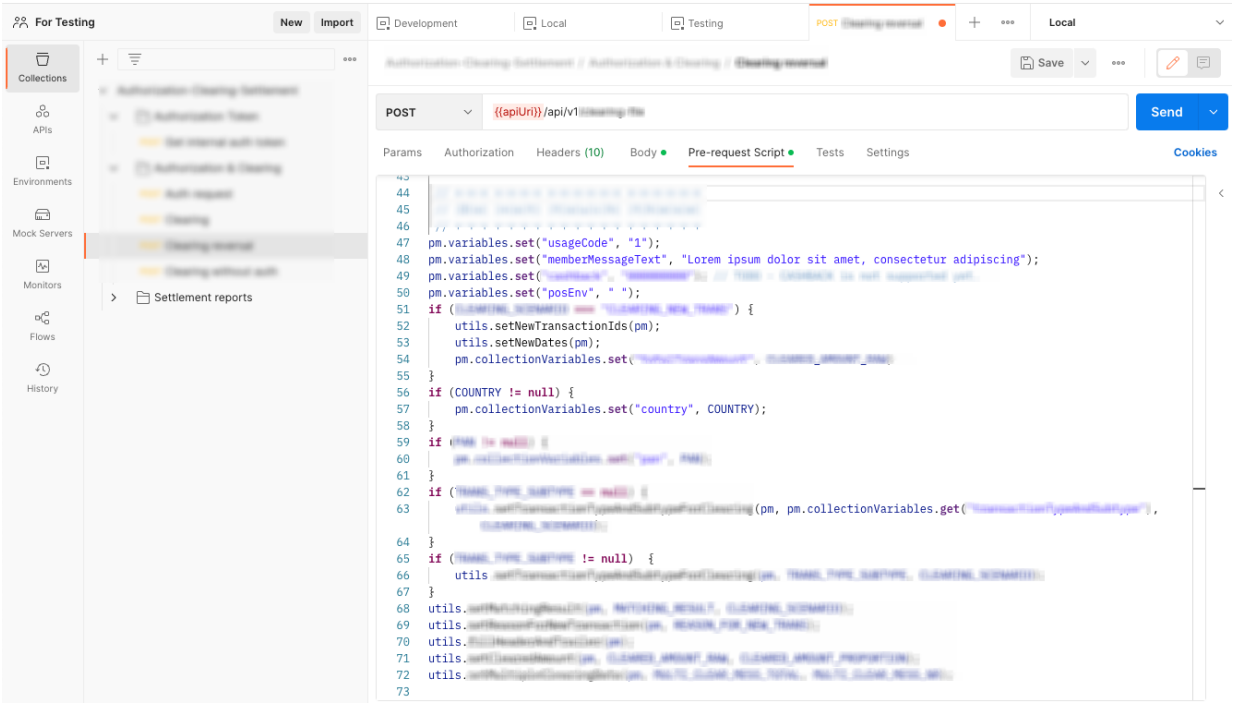
Enne kutsungite tegemist on tarvis end autoriseerida. See on oluline, sest nii on tagatud, et ainult õigustatud kasutajad saavad juurdepääsu teatud ressurssidele. Nii on volitamata juurdepääs piiratud ja andmed turvalised.



Joonis 8 Autoriseerimise vaade.

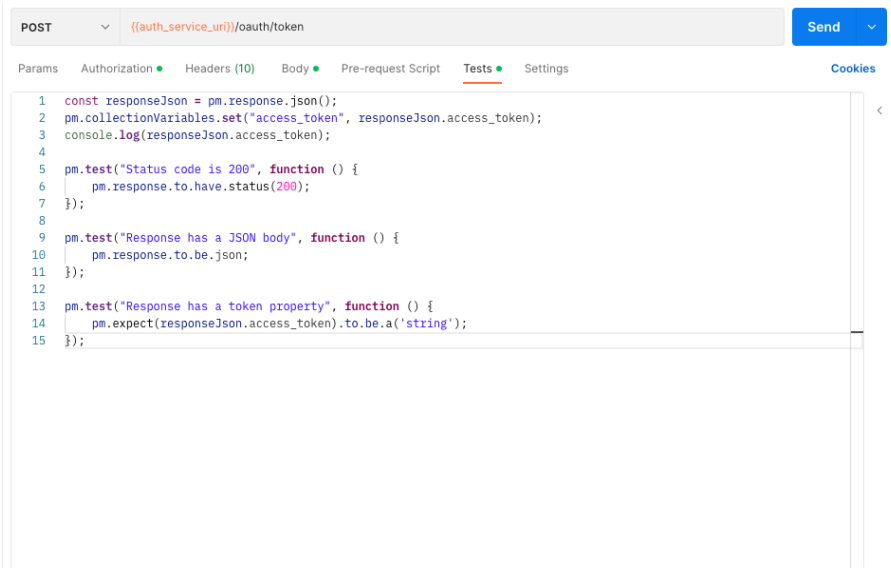
Joonisel on näidatud, kuidas eelnevalt muutujatesse määratud aadress ja kasutaja andmed on sisestatud vajalikesse lahtritesse. Saates õigete parameetritega POST kutsungi antud aadressile, saame vastuseks unikaalse identifikaatori (inglise keeles *token*), mis on justkui pääse süsteemi ning sel on määratud ka kehtivusaeg. Pärast kehtivusaja lõppu tuleb kasutajal end uuesti autoriseerida.

Postmanis on võimalik kõik päringud grupeerida vastavasse kogumisse, mille sees saab töö organiseerida kaustadesse ja alamkaustadesse. Tavapärane praktika näeb välja nii, et iga lõpp-punkti jaoks luuakse eraldi kaust (D. Sekulovski, 2023). Alljärgnevalt jooniselt on näha loodud päringueelne skript (inglise keeles *pre-request script*), mis võimaldab rakendusliidese päringule lisada erinevaid andmeid enne kutsungi tegemist.



Joonis 9 Päringueelse skripti vaade.

"*Tests*" lahtrisse saab kirjutada teste päringu vastuse valideerimiseks. Siin on näide (joonis 10) ühest autoriseerimise kutsungi testist:



Joonis 10 Testid Postmanis.

Kokkuvõttes on Postman kasutajasõbralik ja mugav tööriist rakendusliideste automaattestide kirjutamiseks, millel on väga põhjalik dokumentatsioon.

4.1.2. REST-assured

REST-assured on Java programmeerimiskeele teek, mis on loodud REST tüüpi rakendusliideste automatiseeritud testimiseks ja dokumenteerimiseks. See võimaldab luua lihtsaid ja loetavaid automaatteste Java keeles. REST-assured pakub intuitiivset liidest, mis võimaldab testijatel kirjutada teste kasutades HTTP-protokolli. See on avatud lähtekoodiga, millele on lisatud mitmeid täiendavaid meetodeid ja teisi teeke (T. Hamilton, 2023).

Töö autor kasutab IntelliJ IDEA integreeritud arenduskeskkonda loodud testsüsteemi, millele on lisatud REST-assured teek.

Rakenduse lõpp-punktidele päringute tegemiseks on loodud meetodid, mis kasutavad REST-assured teeki sisseehitatud funktsioone, nagu näiteks "*.given()*", kuhu saab sisendiks anda päringu päised, parameetrid, päringu sisu ja küpsised:

```
24 usages
public Response createPutRequest(String apiPathWithParams, String payloadBody) {
    validateRequestParams(apiPathWithParams, payloadBody, "PUT");
    return given().header("Content-Type", "application/json") RequestSpecification
        .header("Accept", "application/json")
        .header("X-[illegible]-Organization", PropertyLoader.getOrgCode() + "[illegible]")
        .header("Authorization", "Bearer " + authenticationClient.getToken())
        .body(payloadBody)
        .when().put(getServiceHost() + apiPathWithParams) Response
        .then().log().all() ValidatableResponse
        .extract() ExtractableResponse<Response>
        .response();
}

3 usages
private void validateRequestParams(String apiPath, String payloadBody, String requestMethod) {
    if (getServiceHost() == null || getServiceHost().equals("")) {
        throw new RuntimeException("No service host defined. Did you forget to implement getServiceHost()?");
    }

    if (apiPath == null) {
        throw new RuntimeException("Api request path missing. For example [illegible]/customer/create");
    }

    if (payloadBody == null) {
        throw new RuntimeException("Payload cannot be null! Add \"\" if you want to send empty payload.");
    }

    logger.info("Made a request: " + requestMethod + " " + this.getServiceHost() + apiPath);
    logger.info("Payload body: " + (payloadBody.isEmpty() ? "-" : payloadBody) + "\n RESPONSE: ");
}
```

Koodinäide 3 Meetod PUT päringu koostamiseks kasutades REST-assured teeki.

Ülaloleval koodinäitel (koodinäide 3) on näha PUT päringu koostamiseks loodud meetod, mida saab kasutada iga rakendusliidese PUT lõpp-punkti kutsungi tegemisel. Järgneval koodinäitel (koodinäide 4) on näidatud kutsungi tegemist vastu rakenduse "*/api/account*" lõpp-punkti, millele antakse sisendina vajalikud andmed.

```
3 usages
public AccountDataResponse updateAccount(JsonNode payload) {
    return createPutRequest("/api/account", payload.toString()) Response
        .then() ValidatableResponse
        .statusCode(HTTP_STATUS_OK)
        .extract() ExtractableResponse<Response>
        .as(AccountDataResponse.class);
}
```

Koodinäide 4 Näide päringust.

Lõpp-punkti korrektse toimimise valideerimiseks on loodud testjuhtum, kus pannakse kokku vajalikud andmed ning andmekogu saadetakse rakendusliidese lõpp-punktile (koodinäide 5):

```java
@Test
public void createNewAccount(ITestContext context) {
    CustomerPayloadDto customer = (CustomerPayloadDto) context.getAttribute(CUSTOMER.name());
    Long customerId = customer.getCustomerId();
    AccountDto account = new AccountDto(
        customerId,
        AccountTypeCode.getAccountTypeCodeForOrgCode(PropertyLoader.getOrgCode()),
        Currency.getCurrencyFromOrgCode(PropertyLoader.getOrgCode()));
    JsonNode accountPayload = JsonUtil.buildJson(account);
    accountApi.createNewAccount(accountPayload);
    List<AccountResponse> accountResponseList = accountApi.getCustomerAccounts(customerId);
    Assert.assertTrue(accountResponseList.size() > 0);
    Assert.assertEquals(accountResponseList.get(0).getStatusCode(), AccountStatusCode.OPEN);
    Long accountId = accountResponseList.get(0).getId();
    context.setAttribute(ACCOUNT_ID.name(), accountId);
    LOGGER.info("Created account {} for customer {}", accountId, customerId);
}
```

Koodinäide 5 Näide testjuhtumist, kus testitakse arvelduskonto loomise lõpp-punkti.

Kokkuvõttes on REST-assured hea valik Java-põhise testsüsteemi loomisel, kuna pakub testimise tõhusamaks muutmiseks erinevaid meetodeid ja on avatud lähtekoodiga, millel on suur kasutajate kogukond. Kuna töö autori kasutatav testsüsteem on kirjutatud Java programmeerimiskeeles, siis REST-assured on süsteemi edendamiseks igati sobiv valik.

4.2. Testraamistike võrdlus ja valik

Testraamistikud on juhised, protokollid ja standardid, mis annavad aluse testide loomiseks ja käivitamiseks ning aitavad tagada testide järjepidevust ning kordust (D. Atesogullari, A. Mishra, 2020). Testraamistike valikul on samuti arvestatud mitmete teguritega, nende hulgas raamistike populaarsusega ja kindlatele tingimustele vastavusega, näiteks Java keele tugi ja ühilduvus teiste arendus- ja testimistööriistadega. Oluliseks kriteeriumiks oli ka see, et valitud testraamistikul oleks aktiivne toetus ja tugev arendajate kogukond.

Populaarsemate testraamistike hulka kuuluvad (K. Salman, 2023):

- TestNG;
- JUnit;
- Mockito.

4.2.1. Mockito

Mockito on Java-põhine avatud lähtekoodiga testimisraamistik, mis võimaldab testijatel simuleerida teatud klasside, liideste ja objektide käitumist testide ajal. Mockito abil saab testida tarkvara komponente, mis sõltuvad teistest komponentidest, ilma et neid oleks vaja reaalselt käivitada. Mockito võimaldab testijal luua libaobjekte (inglise keeles *mock objects*) ja simuleerida nende käitumist vastavalt testide nõuetele (Javatpoint, 2023).

Mocking on testimistehnika, milles kasutatakse libaobjekte reaalsete süsteemikomponentide simuleerimiseks. See on kasulik tehnika juhul, kui päris tarkvarakomponendid teostavad aeglaseid operatsioone, näiteks andmebaasiühenduste loomine või muid keerukaid lugemis- või kirjutamisfunktsioone. Teinekord võib andmebaasi päringu tegemine võtta aega mitukümmend sekundit, sellisel juhul on mõistlik luua libaobjekt (Javatpoint, 2023).

Järgneval koodinäitel (koodinäide 6) on näide testjuhtumist, kus on kasutatud libaobjekti ning erinevaid Mockito meetodeid:

```
@Test
public void [illegible]() {
    // given
    [illegible] details1 = new [illegible]();
    details1.setCustomerId(1L);

    [illegible] details2 = new [illegible]();
    details2.setCustomerId(1L);

    [illegible] details3 = new [illegible]();
    details3.setCustomerId(3L);

    when([illegible].[illegible](asList(1L, 2L, 3L))).thenReturn(asList(details1, details2, details3));

    // when
    Map<Long, List<[illegible]>> result = service.[illegible](asList(1L, 2L, 3L));

    // then
    assertThat(result).hasSize(2);

    assertThat(result.get(1L)).hasSize(2);
    assertThat(result.get(1L).get(0).getCustomerId()).isEqualTo(1L);
    assertThat(result.get(1L).get(1).getCustomerId()).isEqualTo(1L);
    assertThat(result.get(3L)).hasSize(1);
    assertThat(result.get(3L).get(0).getCustomerId()).isEqualTo(3L);
}
```

Koodinäide 6 Testjuhtum, kus on kasutatud Mockito raamistikku.

4.2.2. JUnit

JUnit on Java keelele loodud avatud lähtekoodiga testraamistik. JUnit sisaldab valmiskirjutatud testklasside teeke, mida saab kasutada erinevate Java klasside testimiseks. Testjuhtumid on kirjutatud meetoditena, millel on spetsiaalsed annotatsioonid (*@Test*) tähistamaks, et need meetotid on testjuhtumid. Testid võivad sisaldada erinevaid aspekte, nagu näiteks testide eelseadistamine (*@Before*) ja toimingud peale testi lõppu (*@After*).

JUnit suudab teste automaatselt käivitada ning tulemusi hinnata, andes hinnangu iga testi õnnestumise või ebaõnnestumise kohta. Seda raamistikku on võimalik ka laiendada, võimaldades kasutajatel lisada teeki täiendavaid funktsioone ja kohandada teste vastavalt vajadustele (JUnit, 2023).

Alloleval koodinäitel (koodinäide 7) on näha *@Before* annotatsiooniga märgitud testmeetod, mis on mõeldud testklassi jaoks vajalike andmete eelseadistuseks ning selline meetod jooksutatakse esimesena.

```
@Before
public void setUp() {
    when(systemPropertyService.requireBoolean([illegible])).thenReturn(true);
    UserContextHolder.setContext(new UserContext("test-user", [illegible], null));

    extractor = new [illegible](
            [illegible],
            [illegible],
            [illegible],
            [illegible],
            [illegible],
            [illegible],
            [illegible],
            [illegible],
            [illegible],
            [illegible],
            [illegible],
            [illegible],
            [illegible],
            [illegible],
            [illegible]()
    );

    ContractHeaderDto headerDto = new ContractHeaderDto();
    headerDto.setLoanTypeCode(ProductTypeCode.[illegible]);
    when([illegible](anyLong())).thenReturn(headerDto);
}
```

Koodinäide 7 JUnit testklassi eelseadistamine.

Mõningal juhul on vaja teha teatuid toiminguid pärast iga testjuhtumi jooksutamist. Koodinäites (koodinäide 8) on näidatud *@After* annotatsiooniga meetodi kasutamist, mis antud näite puhul puhastab mälust info eelnevalt seatud kasutaja andmete kohta:

```
@After
public void tearDown() {
    UserContextHolder.clearContext();
}
```

Koodinäide 2 JUnit @After annotatsiooni kasutamine.

4.2.3. TestNG

TestNG on Java-põhine testraamistik ning sarnaselt JUnit'ile võimaldab see kirjutada testjuhtumeid, mida saab automaatselt käivitada. TestNG on testijate seas populaarne valik, kuna pakub mitmeid funktsioone, mis toetavad erinevat tüüpi testimist ning sellel on võimalus käivitada teste paralleelselt. TestNG pakub JUnit'ist rohkem paindlikkust, mis võimaldab testijatel veelgi enam kohandada testide käitumist vastavalt vajadustele. TestNG pakub toetust ka andmepõhisele testimisele, mis lubab testijatel käivitada sama testjuhtumit erinevate andmetega.

Üks oluline funktsioon on sõltuvustestimine (inglise keeles *dependency testing*), mis võimaldab testide kirjutajal määratleda sõltuvused testjuhtumite vahel. Sellise testimise eesmärgiks on hinnata tarkvara komponentide vahelisi sõltuvusi (TestNG, 2022).

Analoogselt JUnit'ile kasutab ka TestNG annotatsioone tähistamaks teatud samme testklassis. Näide (koodinäide 9) *@BeforeClass* annotatsioonist, mis jooksutatakse kõige esimesena, valmistades ette testklassi jaoks vajalikke andmeid:

```
@BeforeClass
public void beforeTests(ITestContext context) {
    [illegible] customer = [illegible]();
    context.setAttribute(CUSTOMER.name(), customer);
    [illegible] seller = [illegible]();
    context.setAttribute(SELLER.name(), seller);
    [illegible](seller.getCustomerId());
    [illegible] vehicleType = [illegible];
    [illegible] activeDealType = [illegible];
    Long assetId = [illegible](vehicleType, activeDealType);
    context.setAttribute(ASSET.name(), assetId);
    [illegible] leasing = [illegible](assetId, customer, seller);
    context.setAttribute([illegible] name(), leasing);
}
```

Koodinäide 9 TestNG @BeforeClass annotatsioon.

Näide (koodinäide 10) sõltuvuste kasutamisest testklassis, kus testmeetodid sõltuvad eelmiste meetodite tulemustest:

```
3 usages
@Test(dependsOnMethods = { "[illegible]", "[illegible]" })
public void [illegible](ITestContext context) {
    [illegible] loan = ([illegible]) context.getAttribute([illegible].name());
    loanApi.[illegible](loan.get[illegible](), [illegible].getCurrentDate()).then().statusCode(HTTP_STATUS_OK);
    [illegible] loanDetails = loanApi.[illegible]());
    Assert.assertEquals(loanDetails.[illegible]);
}
```

Koodinäide 10 TestNG sõltuvuste kasutamine.

Kokkuvõttes on Java programmeerimiskeelele loodud mitmeid kasulikke testimiseks mõeldud raamistikke, mida on võimalik ka koos kasutada (näiteks Mockito't on võimalik integreerida JUnit'i või TestNG'ga).

4.3. Kirjutatud testide ülevaade ja analüüs

Arvestades eelmistes peatükkides välja toodud testimiseks mõeldud tööriistade ja raamistike plusse ja miinuseid, otsustas töö autor valida olemasoleva testimissüsteemi efektiivsemaks muutmiseks välja järgnevad vahendid - REST-assured ja TestNG. Diplomitöö käigus on valminud mitmed tarkvara rakendusliideste testimiseks loodud regressioonitestid ning käesolevas peatükis on kirjeldatud testimissüsteemi ja loodud teste.

Testimissüsteemi hallatakse IntelliJ IDEA arenduskeskkonnas ning selleks on loodud eraldiseisev repositoorium. Projektis kasutatakse Gradle paketihaldurit ning sõltuvustena on lisatud eelnimetatud REST-assured ja TestNG teegid.

```
test {
    useTestNG() {
        systemProperty "AUTH_URL", System.getProperty("AUTH_URL")
        systemProperty "AUTH_USERNAME", System.getProperty("AUTH_USERNAME")
        systemProperty "AUTH_PASSWORD", System.getProperty("AUTH_PASSWORD")
        listeners << 'ee.TestNgListener'
        useDefaultListeners = true
        reports {
            junitXml.required = false
            html.required = true
            outputDirectory = file("$project.buildDir/TestNGReports/")
        }
        suites 'src/test/resources/ParallelRun.xml'
dependencies {
    def restAssuredVersion : String = '4+'
    implementation group: 'io.rest-assured', name: 'json-path', version: restAssuredVersion
    implementation group: 'io.rest-assured', name: 'json-schema-validator', version: restAssuredVersion
    implementation group: 'io.rest-assured', name: 'rest-assured', version: restAssuredVersion

    implementation group: 'org.testng', name: 'testng', version: '7.5+'
```

Koodinäide 11 Projekti sõltuvused.

Töö autor koostas diplomitöö käigus testimissüsteemile juurde 40 testi, tõstes süsteemi regressioonitestide koguarvu 560 pealt 600 peale. Kui suur protsent kogu rakenduse koodist on regressioonitestidega kaetud on hetkel raske hinnata, kuna rakendusele pole paigaldatud koodianalüüsi tööriista, kuid töö autori arvates on üle poole tarkvarast kaetud.

Kasutades eelnimetatud tööriistu muutus testimine efektiivsemaks, seda väidet toetavad järgnevad näited. Näiteks muutis testimisprotsessi tõhusamaks TestNG *@BeforeClass* annotatsiooni kasutamine, kuna sellega oli mugav testklassi alguses määrata vajalikud ressursid ning neid testjuhtumite sees kasutada. See vähendas koodi korduvust, kuna samu ressursse ei pidanud iga testjuhtumi sees uuesti looma. Tänu REST-assured sisseehitatud funktsioonidele oli rakenduse lõpp-punktidele päringute tegemise ülesseadmine loogiline ja lihtne, mis parandas samuti testimise efektiivsust. Testimistulemuste kontrollimiseks oli hea abivahend TestNG raporteerimise tööriist, mis näitas üksikasjalikult testide õnnestumised ja ebaõnnestumised ning viimase korral ka konkreetse vea pinujälje (inglise keeles *stack trace*). Kuna paljudele rakenduliideste lõpp-punktidele, mis varem nõudsid manuaalset testimist, on

nüüdseks loodud automaattestid, on regressioonitestimise protsess kiirem ning vähenenud on võimalus lohakusvigadele, mis võivad tekkida käsitsi testimisel.

Järgnevatel koodinäidetel (koodinäide 12, 13 ja 14) on näha mõned loodud testjuhtumid:

```
4 usages   taaniel.levin
@Test
public void [redacted](ITestContext context) {
    [redacted] customer = [redacted] context.getAttribute(CUSTOMER.name());
    [redacted] customerRelation = [redacted] context.getAttribute("customerRelation");
    LoanContractDto loanContract = contractBuilderFactory
        .createBuilder()
        [redacted](customer)
        [redacted](customerRelation, [redacted])
        .getLoanContractDto();
    JsonNode loanPayload = JsonUtil.buildJson(loanContract);
    [redacted] loan = loanApi.initializeLoan(loanPayload);
    context.setAttribute([redacted].name(), loan);
    context.setAttribute("loanPeriod", [redacted]);
    context.setAttribute("loanAmount", [redacted]);
    Assert.assertTrue(loan.[redacted] > 0);
    LoggerUtil.logLoanLink(customer.getCustomerId(), loan.[redacted]);
}
```

Koodinäide 12 Testjuhtumi näide, kus testitakse ühte rakendusliidese lõpp-punkti erinevate parameetritega.

```
taaniel.levin
@Test(dependsOnMethods = { "createNewMedicalLoan", "contractPrintoutsAreGenerated", "signMedicalLoan", "activateMedicalLoan" })
public void terminateMedicalLoan(ITestContext context) {
    LoanInitializationResponse loan = (LoanInitializationResponse) context.getAttribute([redacted].name());
    CustomerPayloadDto customer = (CustomerPayloadDto) context.getAttribute(CUSTOMER.name());
    LoanTerminationDto termination = loanContractService.createLoanTerminationDto();
    customerService.updateCustomerForLoanTermination(customer, termination.getReasonCode());
    JsonNode terminationPayload = JsonUtil.buildJson(termination);
    Response terminationResponse = loanApi.terminateContractVersion(loan.getContractVersionId(), terminationPayload);
    terminationResponse.then().statusCode(HTTP_STATUS_OK);
    ContractDetailsResponse detailsResponse = loanApi.getContractDetails(loan.getContractHeaderId());
    Assert.assertEquals([redacted], ContractStatus.ACTIVE_TERM);
}
```

Koodinäide 13 Testjuhtumi näide, kus testitakse rakendusliidese lõpp-punkti.

```
taaniel.levin
@Test(dependsOnMethods = {
    "createNewInvestementLoan",
    "generateContractPrintoutsForInvestementLoan",
    "signInvestementLoan",
    "activateInvestementLoan"
})
public void investmentLoanPayout(ITestContext context) throws IOException {
    CustomerPayloadDto customer = (CustomerPayloadDto) context.getAttribute(CUSTOMER.name());
    LoanInitializationResponse loan = (LoanInitializationResponse) context.getAttribute(INVESTMENT_LOAN.name());
    AccountDataResponse accountData = (AccountDataResponse) context.getAttribute("accountData");
    BigDecimal payoutAmount = new BigDecimal(String.valueOf(context.getAttribute("loanAmount")));
    String referenceNumber = loanApi.getPaymentInformationLoan.getContractHeaderId().getReferenceNumber();
    LoanPayoutDto loanPayoutDto = new LoanPayoutDto(
        AccountOwner.CUSTOMER_SERVICING_ACCOUNT,
        accountData.getIban(),
        "Mr. Money",
        Currency.EUR,
        "Payout",
        customer.getCustomerId(),
        referenceNumber,
        PaymentInstructionStatusCode.SEND_TO_CONFIRMATION);
    loanPayoutDto.setAmount(payoutAmount);
    loanPayoutDto.setDisbursementFee(BigDecimal.ZERO);
    JsonNode payoutPayload = JsonUtil.buildJson(loanPayoutDto);

    loanApi.createLoanPayout(loan.getContractHeaderId(), payoutPayload) Response
        .then() ValidatableResponse
        .statusCode(HTTP_STATUS_NO_CONTENT);

    List<PaymentInstructionDto> paymentInstructions = loanApi.getPaymentInstructions(loan.getContractHeaderId());
    PaymentInstructionDto latestPaymentInstruction = paymentInstructions
        .stream() Stream<PaymentInstructionDto>
        .max(Comparator.comparing(PaymentInstructionDto::getId)) Optional<PaymentInstructionDto>
        .orElse(null);
    Assert.assertNotNull(latestPaymentInstruction);

    loanContractService.acceptPayout(loan, latestPaymentInstruction.getId());

    ContractDetailsResponse loanDetails = loanApi.getContractDetails(loan.getContractHeaderId());
    int loanPeriod = (int) context.getAttribute("loanPeriod");
    Assert.assertEquals(loanDetails.getNumberOfScheduleRows(), Long.valueOf(loanPeriod));
}
```

Koodinäide 14 Testjuhtumi näide, kus ühes pikas testjuhtumis testitakse erinevaid lõpp-punkte.

KOKKUVÕTE

Töö autori töökohal kasutatavas tarkvaras esineb palju vigu, kuna kasutusel olev automaattestimissüsteem on ebaefektiivne. Käesoleva diplomitöö eesmärk oli töötada välja efektiivsem rakendusliideste regressioonitestimise tarbeks mõeldud testimissüsteem autori tööandja näitel.

Eesmärgi saavutamiseks võrreldi erinevate kriteeriumite alusel testimiseks mõeldud tööriistu ja raamistikke ning valitud vahenditega loodi testimissüsteemi tarbeks uusi testjuhtumeid. Võrdluse käigus saadi teada mitmetest funktsionaalsustest, mida erinevad testimistööriistad ja -raamistikud pakuvad ning läbi mille oli võimalik testimist efektiivsemaks muuta.

Diplomitöö eesmärk sai täidetud ning olemasolevat testimissüsteemi täiustati 40 uue testjuhtumiga. Valitud testimisvahendid muutsid testimissüsteemi kasutamist tõhusamaks ning ka testimistulemuste analüüs oli selgem. Edaspidi on uute testjuhtumite kirjutamine seega lihtsam ja kiirem, mis aitab kokku hoida olulist ressurssi.

Diplomitöö on kasulik neile, kes soovivad samuti regressiooniteste automatiseerida, kuna töö eeskujul on võimalik ka teistel inimestel ja organisatsioonidel koostada tõhus automaattestimissüsteem. Järgmise sammuna tuleks testimissüsteemi veelgi täiendada lisades juurde testjuhtumeid. Eesmärgiks võiks olla kogu tarkvara koodi katmine testidega, et tarkvara oleks võimalikult kvaliteetne ja vead leitaks varajases staadiumis.

SUMMARY

Title in English: Automating Regression Testing for Application Programming Interfaces

The software used at the author's workplace suffers from numerous bugs due to the inefficiency of the existing automated testing system. The objective of this thesis was to develop a more efficient testing system for regression testing of application interfaces, using the author's employer as a case study.

To achieve this objective, various testing tools and frameworks were compared based on different criteria and new test cases were created using the selected tools for the testing system. The comparison revealed several functionalities offered by different testing tools and frameworks, through which testing efficiency was improved.

The objective of the thesis was successfully accomplished and the existing testing system was enhanced with numerous new test cases. The selected testing tools made the use of the testing system more efficient and the analysis of test results became clearer. Consequently, writing new test cases in the future will be easier and faster, resulting in significant resource savings.

This thesis is valuable for individuals and organizations interested in automating regression testing, as it provides an example for developing an effective automated testing system. The next step should involve further enhancements to the testing system by adding more test cases. The ultimate goal should be to achieve comprehensive code coverage with tests, ensuring high-quality software and early detection of defects.

ALLIKAD

Software Testing Help. (2022). *Regression Testing Tools and Methods* [25. november, 2022]. https://www.softwaretestinghelp.com/regression-testing-tools-and-methods/

E. Engström. (2010). *Exploring Regression Testing and Software Product Line Testing Research and State of Practice.* [25. november, 2022]. https://lucris.lub.lu.se/ws/portalfiles/portal/5257989/1604065.pdf

A. Rahmani, J. L. Min, A. Maspupah. (2021). *An empirical study of regression testing techniques.* [25. november, 2022]. https://iopscience.iop.org/article/10.1088/1742-6596/1869/1/012080/pdf

A. Ehsan, M. A. M. E. Abuhaliqa, C. Catal, D. Mishra. (2022). *RESTful API Testing Methodologies: Rationale, Challenges, and Solution Directions.* [25. november, 2022]. https://www.mdpi.com/2076-3417/12/9/4369/htm

S. Ambler. (2012). *Agile Testing and Quality Strategies: Discipline Over Rhetoric.*

[25. november, 2022]. http://www.ambysoft.com/essays/agileTesting.html

T. Hamilton. (2022). *What is software testing?* [15. detsember, 2022].

https://www.guru99.com/software-testing-introduction-importance.html

J. Varun Iyer. (2023). *Different types of API protocols.* [10. aprill, 2023].

https://iq.opengenus.org/different-types-of-api-protocols/

Isha, A. Sharma, M. Revathi. (2018). *Automated API Testing.* [11. aprill, 2023].

https://ieeexplore.ieee.org/abstract/document/9034254

T. Hamilton. (2023). *Software Testing Methodologies: QA Models.* [11. aprill, 2023].

https://www.guru99.com/testing-methodology.html

Software Testing Fundamentals. (2023). *Unit Testing.* [11. aprill, 2023]. https://softwaretestingfundamentals.com/unit-testing/

International Software Test Institute. (2023). *Software Testing Levels.* [12 .aprill, 2023]. https://www.test-institute.org/Software_Testing_Levels.php

Software Testing Help. (2023). *Regression Testing Tools and Methods.* [12. aprill, 2023]. https://www.softwaretestinghelp.com/regression-testing-tools-and-methods/

Testsigma. (2023). *A Detailed analysis on Advantages, Disadvantages, Challenges and Risks of Regression Testing*. [12. aprill, 2023]. https://testsigma.com/regression-testing/advantages-of-regression-testing

M. Pezzè, M. Young. (2008). *Software Testing and Analysis: Process, Principles, and Techniques.* [24. aprill, 2023]. https://ix.cs.uoregon.edu/~michal/book/Samples/book.pdf

A. Walker. (2023). *What is an API? Full Form, Meaning, Definition, Types & Example.* [24. aprill, 2023]. https://www.guru99.com/what-is-api.html

W3Schools. (2003). [24. aprill, 2023]. https://www.w3schools.com/xml/xml_soap.asp

M. A. Umar. (2019). *Comprehensive study of software testing: Categories, levels, techniques, and types*. [24. aprill, 2023]. https://www.researchgate.net/publication/337331361_Comprehensive_study_of_software_testing_Categories_levels_techniques_and_types

D. Sekulovski. (2023). *Using Postman for API Testing.* [25. aprill, 2023]. https://www.testdevlab.com/blog/using-postman-for-api-testing

T. Hamilton. (2023). *REST Assured Tutorial for API Automation Testing.* [25. aprill, 2023]. https://www.guru99.com/rest-assured.html

Javatpoint. (2023). *Mockito Framework Tutorial.* [26. aprill, 2023]. https://www.javatpoint.com/mockito

D. Atesogullari, A. Mishra. (2020). *Automation Testing Tools: A Comparative View.* [26. aprill, 2023]. https://www.researchgate.net/publication/346109409_AUTOMATION_TESTING_TOOLS_A_COMPARATIVE_VIEW

JUnit. (2023). [27. aprill, 2023]. https://junit.org/junit5/docs/current/user-guide/junit-user-guide-5.9.3.pdf

TestNG. (2022). [27. aprill, 2023]. https://testng.org/doc/documentation-main.html

Scrum Alliance. (2023). *Everything You Need to Know About Acceptance Criteria.* [15. mai, 2023]. https://resources.scrumalliance.org/Article/need-know-acceptance-criteria

Software Testing Material. (2023). *10 Best API Testing Tools In 2023.* [16. mai, 2023]. https://www.softwaretestingmaterial.com/best-api-testing-tools/

K. Salman. (2023). *13 Best Java Testing Frameworks For 2023.* [16. mai, 2023]. https://www.lambdatest.com/blog/best-java-testing-frameworks/